# **№ 11 Реализация MVVM на WPF**

## Задание

1. Разобраться с примером выдачи/приема книг в библиотеке, построенном на основе шаблона MVVM (Model-View-ViewModel) (в папке SampleMVVM). Изучить архитектуру и принцип построения шаблона.
2. Разработать по аналогии приложение WPF, построенное на основе MVVM. Для хранения использовать базу данных.

|  |  |
| --- | --- |
| 1, 5, 13 | Медцентр  В поликлинике существуют определенные отделения, категории врачей, специализации. В окне выполнить поиск врача с просмотром информации: ФИО, специальность, фото т.д. Организовать выдачу/отмену талонов. |
| 2, 6, 10, 14 | Курсы  В образовательном центре существует набор дисциплин: название, количество часов, лектор, количество слушателей т.д. Организовать зачисление студентов на дисциплину: запись/отмену на курс определенного лектора. |
| 3, 11, 15 | Магазин  В магазине имеется определенная продукция: наименование, цена, количество и т.п. Организовать добавление и удаление товаров в корзину. |
| 4, 8, 16 | Журнал  На факультете есть специальности, группы, подгруппы, курсы. Организовать выставление оценок и пропусков по дисциплине ООП. |
| 7, 9, 12 | Запись на консультацию  У преподавателя есть выделенное время для консультаций и сдачи: фио, предмет, дата, временной интервал. Организовать запись и отмену на консультацию . |
| *Дополнительное задание (по желанию)* | *Сделайте управление (свойства, размеры, количество и тп.) игровыми объектами (птицы, колобки, самолеты, танки), которые меняют свое визуальное отображение или движение.* |

## Вопросы

1. Что такое MVC, MVP, MVVM?
2. Объясните из каких компонентов состоит паттерн MVVM?
3. Каким образом паттерн можно реализовать в WPF?
4. Объясните назначение интерфейса INotifyPropertyChanged?
5. Зачем нужен интерфейс ICommand и как паттерн MVVM использует команды в WPF-приложениях.
6. Чем отличаются команды от событий?
7. Расскажите о назначении и об основных возможностях следующих библиотек и фреймворков: Light Toolkit, Catel и Prism.

## Теоретические сведения:

MVVM – Model – View – ViewModel – паттерн организации PL (**presentation layer** – уровень представления).

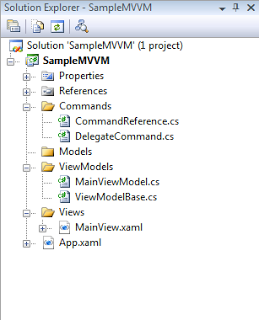
Паттерн MVVM применяется при создании приложений с помощью WPF.   Этот паттерн был придуман - John Gossman. Идеологически MVVM похож на [Presentation Model](http://martinfowler.com/eaaDev/PresentationModel.html) описанный Фаулером, но MVVM сильно опирается на возможности WPF.

Шаблон MVVM имеет три основных компонента: *модель*, которая представляет бизнес-логику приложения, *представление* пользовательского интерфейса XAML, и *представление-модель*, в котором содержится вся логика построения графического интерфейса и ссылка на модель, поэтому он выступает в качестве модели для представления

Основная особенность MVVM заключается в том, что все поведение выносится из **представления (view)** в**модель представления (view model)**.  Связывание представления и модели представления осуществляется декларативными байндингами в XAML разметке. Это позволяет тестировать все детали интерфейса не используя сложных инструментальных средств.

Рассмотрим проект.

Для проекта, построенного на основе MVVM создается следующая структура файлов:

[](http://1.bp.blogspot.com/_j6rWiCD_MnA/S78eK8P4P_I/AAAAAAAABVQ/2_4YwHTFePU/s1600/mvvm_files.png)

Пусть надо реализовать следующую задачу: отображение списка книг читального зала. У книги есть: Название, Автор, Доступное количество

Пусть ходят читатели берут книги почитать или возвращают их обратно. Надо в любой момент знать, сколько экземпляров той или иной книги осталось и можем ли мы ее выдать. На примере задачи рассмотрим MVVM

**Model**

Model — это сущности системы. Модель будет состоять из одного простого класса:

# Book.cs

class Book

{

public string Title { get; set; }

public string Author { get; set; }

public int Count { get; set; }

public Book(string title, string author, int count)

{

this.Title = title;

this.Author = author;

this.Count = count;

}

}

**ViewModel**

ViewModel —пожалуй, ключевой момент. Это такие специальные классы, которые:

* Осуществляют связь между моделью и формой.
* Отслеживают изменения в данных, произведенные пользователем.
* Обрабатывают логику работы View (механизм команд)

В учетом датабиндинга в WPF это дает следующий результат: в C#-коде формы становится совсем не надо ничего писать. ViewModel для модели:

# BookViewModel.cs

class BookViewModel : ViewModelBase

{

public Book Book;

public BookViewModel(Book book)

{

this.Book = book;

}

public string Title

{

get { return Book.Title; }

set

{

Book.Title = value;

OnPropertyChanged("Title");

}

}

public string Author

{

get { return Book.Author; }

set

{

Book.Author = value;

OnPropertyChanged("Author");

}

}

public int Count

{

get { return Book.Count; }

set

{

Book.Count = value;

OnPropertyChanged("Count");

}

}

BookViewModel унаследован от класса ViewModelBase. ViewModelBase, в свою очередь, реализует интерфейс INotifyPropertyChanged и содержит функцию OnPropertyChanged. Это нужно для того, чтобы всегда можно было вызвать событие "изменилось такое-то поле". Как видно в коде, при любом изменении поля такое событие вызываем и передаем в качестве параметра его название. Потом на форме биндинг может это событие обработать и, как следствие, интерфейс и ViewModel всегда будут друг с другом синхронизированы.

Помимо BookViewModel есть еще класс MainViewModel, связанный с формой. Добавим в него поле:

public ObservableCollection<BookViewModel> BooksList { get; set; }

ObservableCollection — это специальная коллекция, которая умеет отслеживать изменения в себе. Также изменим конструктор:

public MainViewModel(List<Book> books)

{

BooksList = new ObservableCollection<BookViewModel>(books.Select(b => new BookViewModel(b)));

}

**View**

Это окно, либо User Control. У любого FrameworkElement-а WPF есть такое поле DataContext. DataContext может быть любым object-ом, иметь какие угодно поля, а его главная задача — являться источником данных для Databinding-а. Форма всего одна, DataContext для нее заполняется в методе OnStartup, что в App.xaml.cs. Модифицируем и получится следующее:

# App.xaml.cs

public partial class App : Application

{

private void OnStartup(object sender, StartupEventArgs e)

{

List<Book> books = new List<Book>()

{

new Book("Пттерны проетирования", "John Gossman", 3),

new Book("CLR via C#", "Джеффри Рихтер", 2),

new Book("Исскуство программирования", "Кнут", 2)

};

MainView view = new MainView(); // создали View

MainViewModel viewModel = new ViewModels.MainViewModel(books); // Создали ViewModel

view.DataContext = viewModel; // положили ViewModel во View в качестве DataContext

view.Show();

}

}

Осталось написать XAML-код формы:

# MainView.xaml

<ListView ItemsSource="{Binding BooksList}" IsSynchronizedWithCurrentItem="True">

<ListView.ItemTemplate>

<DataTemplate>

<Border BorderBrush="Bisque" BorderThickness="1" Margin="10">

<StackPanel Margin="10">

<TextBlock Text="{Binding Title}" FontWeight="Bold"/>

<TextBlock Text="{Binding Author}" />

<StackPanel Orientation="Horizontal">

<TextBlock Text="Осталось:" />

<TextBlock Text="{Binding Count, Mode=TwoWay}" FontWeight="Bold" Margin="10,0"/>

<TextBlock Text="шт" />

</StackPanel>

</StackPanel>

</Border>

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

Обратите внимание на конструкцию Binding в разметке формы. Таким образом можно привязывать поля объекта, находящегося в DataContext-е, к атрибутам контролов.

**Редактирование**

Для выделенной в списке книги будет открываться редактор. Изменим XAML-разметку формы:

# MainView.xaml

<ListView ItemsSource="{Binding BooksList}" IsSynchronizedWithCurrentItem="True">

<ListView.ItemTemplate>

<DataTemplate>

<Border BorderBrush="Bisque" BorderThickness="1" Margin="10">

<StackPanel Margin="10">

<TextBlock Text="{Binding Title}" FontWeight="Bold"/>

<TextBlock Text="{Binding Author}" />

<StackPanel Orientation="Horizontal">

<TextBlock Text="Осталось:" />

<TextBlock Text="{Binding Count, Mode=TwoWay}" FontWeight="Bold" Margin="10,0"/>

<TextBlock Text="шт" />

</StackPanel>

</StackPanel>

</Border>

</DataTemplate>

</ListView.ItemTemplate>

</ListView>

<ContentControl Grid.Column="1" Content="{Binding BooksList}">

<ContentControl.ContentTemplate>

<DataTemplate>

<Border BorderBrush="Bisque" BorderThickness="1" Margin="10">

<StackPanel Margin="10">

<TextBlock Text="Название:" />

<TextBox Text="{Binding Title, UpdateSourceTrigger=PropertyChanged}" Margin="0,0,0,10"/>

<TextBlock Text="Автор:"/>

<TextBox Text="{Binding Author, UpdateSourceTrigger=PropertyChanged}" Margin="0,0,0,10"/>

<StackPanel Orientation="Horizontal">

<Button Content="Выдать" Command="{Binding GiveItemCommand}" Margin="10,0" />

<Button Content="Забрать" Command="{Binding GetItemCommand}" Margin="10,0" />

</StackPanel>

</StackPanel>

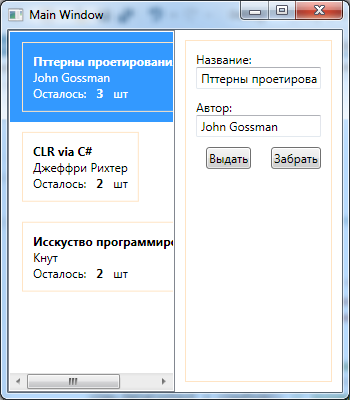
</Border>

</DataTemplate>

</ContentControl.ContentTemplate>

</ContentControl>

 UpdateSourceTrigger=PropertyChanged в строке биндинга. Это значит, что любое изменение, производимое в данном поле, будет немедленно отражаться на источнике:



Если этого не написать, источник будет обновляться только по окончании редактирования (т.е. когда контрол будет терять фокус). Это может привести к следующей ошибке интерфейса: когда нажимаешь "Сохранить", сохраняется все, кроме только что измененного поля.

**Команды**

Пусть некие читатели берут книги и возвращают. Соответственно, сделаем две кнопки — «Выдать» и «Забрать», меняющие количество имеющихся в наличии книг. Если книг не осталось (Count = 0), кнопка «Выдать» должна быть неактивной.

В MVVM не пишутся обработчики событий. Функции, которые нужно выполнять контролам, пишутся во ViewModel и биндятся к контролам точно так же, как поля. Только используется механизм команд.

Команда должна представлять из себя экземпляр класса, реализующего интерфейс ICommand. DelegateCommand – используется для реализации команды без параметров и DelegateCommand<T> — для реализации команды с параметром типа T.

Параметры передавать не будем. Код ViewModel:

# BookViewModel.cs

#region Commands

#region Забрать

private DelegateCommand getItemCommand;

public ICommand GetItemCommand

{

get

{

if (getItemCommand == null)

{

getItemCommand = new DelegateCommand(GetItem);

}

return getItemCommand;

}

}

private void GetItem()

{

Count++;

}

#endregion

#region Выдать

private DelegateCommand giveItemCommand;

public ICommand GiveItemCommand

{

get

{

if (giveItemCommand == null)

{

giveItemCommand = new DelegateCommand(GiveItem, CanGiveItem);

}

return giveItemCommand;

}

}

private void GiveItem()

{

Count--;

}

private bool CanGiveItem()

{

return Count > 0;

}

#endregion

#endregion

}

Код добавляется в BookViewModel, а не в MainViewMode. Будем добавлять кнопки в ContentControl, DataContext-ом которого является именно BookViewModel.

С первой командой - создали команду, и в назначили ей в качестве действия метод GetItem, который и будет вызываться при ее активации. Со второй немного интереснее, но тоже просто. Помимо того, что она выполняет некоторое действие, она еще и может проверять с помощью метода CanGiveItem(), может она выполняться или нет.

В XAML-разметку добавим следующее

# MainView.xaml

<ContentControl Grid.Column="1" Content="{Binding BooksList}">

<ContentControl.ContentTemplate>

<DataTemplate>

<Border BorderBrush="Bisque" BorderThickness="1" Margin="10">

<StackPanel Margin="10">

<TextBlock Text="Название:" />

<TextBox Text="{Binding Title, UpdateSourceTrigger=PropertyChanged}" Margin="0,0,0,10"/>

<TextBlock Text="Автор:"/>

<TextBox Text="{Binding Author, UpdateSourceTrigger=PropertyChanged}" Margin="0,0,0,10"/>

<StackPanel Orientation="Horizontal">

<Button Content="Выдать" Command="{Binding GiveItemCommand}" Margin="10,0" />

<Button Content="Забрать" Command="{Binding GetItemCommand}" Margin="10,0" />

</StackPanel>

</StackPanel>

</Border>

</DataTemplate>

</ContentControl.ContentTemplate>

</ContentControl>

Мы получили требуемую функциональность. Количество экземпляров книги увеличивается и уменьшается, а когда их становится 0, кнопка «Выдать» дизаблится (благодаря упомянутому CanGiveItem).